CHAPTER

FIVE

MODELS, LANGUAGES, AND HEURISTICS

The next three parts form the heart of this book. In these parts we discuss models, languages, and heuristics for coordinated computing. This chapter lays the groundwork for understanding the various systems, pointing out the critical choices involved in system design. We describe the models in Part 2, the languages in Part 3, and the heuristic systems in Part 4. In Part 5, we summarize our observations, comparing and contrasting the approaches taken by the different systems.

This chapter has three sections. We begin by discussing the differences between programming languages, models of programming systems, and heuristic organizations for programming. Models are used to understand and describe computation, while languages are used to command computers. Often a proposed system has some characteristics of each. Heuristics are organizational frameworks for controlling distributed systems.

Coordinated computing is a field of our invention. The authors of the various proposals wrote papers on subjects such as programming languages, the mathematical theory of computation, and artificial intelligence, not expecting to have these papers packaged together with such distant conceptual cousins. In fact, the systems are very different. This divergence arises primarily because the various proposals address different problem domains. Different domains have different key problems and assumptions; the given of one field is often the critical issue of another. In Section 5-2 we discuss the problem domain "soil" in which these proposals grew, listing the assumptions and issues of each intellectual region.

Designing a programming system involves making many decisions. Some of these decisions are uninteresting: Should the assignment symbol be "=" or ":="? We ignore such issues. On the other hand, some decisions are more important: Should assignment be allowed at all? We have identified 12 crucial dimensions for the language/model designer. We introduce these design dimensions in Sections 5-2 and 5-3. Each of the systems in Parts 2, 3, and 4 chooses one or more of the possibilities for each dimension; the reader should be able to recognize the choices. These decisions determine a large part of the structure of a language or model. In Part 5 we compare the choices used by the various systems. Their variety can then be understood to be a selection in the space of critical choices.

5-1 MODELS, LANGUAGES, AND HEURISTICS

Models are used to explain and analyze the behavior of complex systems. A model abstracts the salient properties of a system. Models of concurrent systems usually specify the interprocess communication and synchronization mechanisms. The model is then used to derive properties and predict the behavior of the system. For example, a model of the communication patterns of a set of processes can be used to analyze algorithmic efficiency; a model of the information transfer between processes can be used to prove algorithmic correctness.

Programming languages are used to provide exact directions to computers. In Chapter 2 we observed that programming languages are characterized by their syntax and semantics. The syntax describes the surface appearance of a language, while the semantics is the set of actions that can be effected. The key issue in programming language design is not syntax but semantics. The semantics of a programming language reveals the choice of ontology (set of things that exist) and the set of things that can be done with them—in some sense, computational metaphysics.

The formal semantics of concurrent languages is a complicated subject. Concurrent languages are almost always more powerful than Turing machines. This is because Turing machines are deterministic—they always produce the same output for the same input. Concurrent systems can take advantage of asynchronous processes to produce many different answers for the same input. Similarly, there are differences in operational semantic power among the concurrent languages. Some systems provide primitives for synchronization and timing that are difficult or impossible to imitate as nonprimitives. Except for the presence of such primitives, usually (but not always) the behavior exhibited by one system can be obtained by the other systems. However (like conventional languages), the equivalent of a short program in one language may be a long program in another. In Section 19-3, we discuss a model that can describe the behavior of all the systems in Parts 2, 3, and 4.

Some of our systems are model-language hybrids. A *hybrid* has some languagelike features (typically the critical ones for distributed computing) often with a minimal syntax. The language/model designer then glosses over the remainder of the proposed language, asserting that it is a standard sort of system. The archetypical hybrid adds a few programming constructs to any of a class of languages. This is a good way to express new programming concepts. The syntax and semantics of ordinary programming languages are well understood. By concentrating only on the extensions for distribution, hybrid designers focus on the critical issues.

Heuristic systems are proposed organizations for distributed problem solving. Megacomputing presents both a challenge and an opportunity to programming. On one hand, large distributed systems will provide almost unlimited processing. On the other hand, it may be difficult to organize such a system to actually get anything useful done. Heuristic organizations are often based on taking a particular algorithm or metaphor and projecting it into a system organization. A simple example is taking the idea of committing or terminating a group of actions simultaneously, allowing a programmer to be sure of a distributed consistency. A more grandiose heuristic organization would be a programming system built around the theme: "the agents of a distributed system are individuals in a *laissez faire* economy, buying and selling goods and services to reach their individual goals." Such a system would provide agents, goods (perhaps computing cycles and memory allocations), services (the solutions of subtasks), and currency and a market to structure their interactions. A programmer who can express a task in terms of these objects would find that task easily programmed in such a megacomputing system.

5-2 PROBLEM DOMAINS

The systems we consider in the next three parts are a diverse group. One might be curious about the origin of their variety. Some of this variety arises from the natural inclination of people to find different ways of doing the same thing. However, most of the differences have a more fundamental source—the plethora of different mechanisms arises from a plethora of different problem domains. That is, the designers of these systems are building tools to solve different classes of problems. Thus, each invents a different set of tools. We have included this variety of systems because we believe that these problem domains and solutions are relevant to coordinated computing.

Problem Domain The system designers have their own perspective on the problems of distributed computing and have designed their systems to address just those issues. We identify five major perspectives in these proposals: operating systems, pragmatics, semantics, analysis, and distributed problem solving.

Each system focuses on some combination of these. Often a construct that is meant to solve the problems of one domain implies a set of unacceptable assumptions for another. For example, an elaborate algorithm for choosing processes to communicate is inappropriate for a system directed at implementing underlying communication mechanisms. On the other hand, a system that provides no communication control may be too primitive for describing complex problem-solving architectures.

Some languages address the immediate control of the distributed system. We call this the *operating systems* approach. Such systems emphasize matching the constructs of the distributed language to the physical machines. Key issues in the operating systems approach are the ease of effecting this control and the efficiency of the resulting system. Operating systems approaches sometimes treat processes and processors as synonymous. In such systems communication structure—which processes can communicate with which other processes — often remains static throughout program execution. We frequently use the synchronization problems discussed in Chapter 3 to illustrate operating systems languages.

The *pragmatic* approach is concerned with providing tools to aid the programming process. Pragmatic systems emphasize ease of program expression. Pragmatic languages often include constructs that are not necessarily easy to implement but that the designers feel are important for structuring or aiding programming. Typical examples for pragmatic systems are difficult algorithms simplified by the special constructs of the language.

Some researchers study concurrent computing systems as mathematical objects, hoping to clarify the semantics of concurrent computation and to prove the correctness of programs. We call this perspective the *semantic* approach to coordinated computing. From this perspective, the key metric for appraising a proposal is mathematical elegance. (Of course, a connection to practice provides a sound basis for theoretical work. Thus, the generality of the selected model and its correspondence to elements of real systems are also important.) The proposer of a semantic model displays its virtues with proofs. However, such proofs are beyond our present scope; we remain aware of mathematical grace without directly experiencing it.

Models developed from the *analytic* perspective are concerned with analyzing algorithmic efficiency. The ease of performing that analysis is an important criterion for success of such systems. Like semantic models, analytic models require a close correspondence between the model and the system being modeled.

Some systems are *heuristic organizations* for problem solving. These systems seek to harness concurrent computing systems to work together on difficult symbolic tasks. The natural expression of problems and problem domains and the translation of this expression into efficient distributed programs are the goals of heuristic approaches.

Often a proposal addresses several of these problem areas simultaneously. For example, a pragmatic approach might assert that program verification is an important tool; programs in "good" languages must be easy to prove correct. Similarly, languages for building real systems should have at least a peripheral concern for programming pragmatics.

5-3 PRIMITIVES FOR DISTRIBUTED COMPUTING

Our survey of systems reveals several common themes. We have already examined one such theme, the problem-domain perspective. In this section, we indicate several other key choices for designers of coordinated systems. The description of each system (in Parts 2, 3, and 4) shows how it approaches each dimension.

Explicit Processes The primary characteristic of a coordinated computing system is the simultaneous activity of many computing agents. The first decision for the designer of a coordinated model or language is whether to make the user (programmer) aware of the existence of these agents or to conceal the systems's underlying parallel nature. Systems that provide such agents to the user level have *explicit processes.* Typically, processes have program and storage. Often, processes have names (or addresses). Every system with explicit processes provides some mechanisms for interprocess communication; some explicit process systems treat processes as allocatable objects that can be created (and destroyed) during program execution.

Alternatively, a system can be organized around implicit processes. With implicit processes, computation is performed on request. The user does not specify "who" is to satisfy the request. The theme of these systems is that the user defines what is to be done and the system arranges for its concurrent computation. In general, a part of a system that can accomplish computation is an *agent.* Processes are examples of agents.

Almost all the systems in Parts 2, 3, and 4 use explicit processes. Many of the remaining dimensions of coordinated language and model design deal with interprocess communication and control. Often these issues must be interpreted differently for implicit-process systems.

Process Dynamics In a system with explicit processes, the set of processes can be fixed for the lifetime of the program or the system can allow processes to be created and destroyed during program execution. We say that a system that allows the creation of new processes during program execution supports *dynamic process creation.* A system that treats the set of processes as fixed has *static process allocation.*

Systems with dynamic process creation usually create processes in one of two ways—either by explicitly allocating new processes in an executable statement (comparable to the **new** statement for storage allocation in Pascal), or by the lexical expansion of program text. That is, if process P declares (as one of its variables, so to speak) process Q, then creating a new copy of P lexically creates a new copy of Q.

Many systems give names to newly created processes. These names can be passed between processes; communications can be addressed to processes by their names. Systems with static processes sometimes require that the system be able to determine the interprocess communication structure ("who talks to whom") before program execution ("at compile time").

Systems that allow dynamic process creation usually provide dynamic process destruction. A process that has finished executing has *terminated.* Processes can terminate in several different ways. Almost all systems allow processes to terminate by completing their program. Some systems have more elaborate schemes for process termination, including mechanisms that allow some processes to terminate other processes. One alternative to explicit termination is garbage collection of the resources of inaccessible or useless processes.

Synchronization Those systems that do not have explicit processes communicate through shared storage. Some systems with explicit processes also allow shared storage for interprocess communication. Communication without shared storage is *message* communication.

The two kinds of message transmissions are synchronous and asynchronous messages. The sender of an *asynchronously* transmitted message initiates the message and is then free to continue computing. These are *send-and-forget* communications. With *synchronously* transmitted messages, the communicating parties both attend to the communication. A process that starts a synchronous transmission waits until the message has been both received and acknowledged. We say that a process that is waiting for a synchronous communication to be accepted is *blocked.*

Synchronous communication resembles a procedure call since the caller transfers control and waits until the called agent returns an acknowledgment.* Asynchronous communication is uncommon in conventional programming languages. Metaphorically, synchronous communication can be compared to a telephone call—it requires the attention of both communicators and allows two-way conversations. Asynchronous communication is like mailing a letter—one is not idle until a dispatched letter is delivered, but there is no direct mechanism for achieving an immediate response. Communications are generally requests. A *request* is an attempt by one agent in a computing system to elicit a particular behavior from another.

Whether synchronous or asynchronous communications provide a better structure is a longstanding issue among operating systems designers. In a controversial paper, Lauer and Needham [Lauer 78] argue that (at least for operating

* The term *remote procedure call* has been used to describe the concept of calling a procedure on another machine. Nelson's dissertation [Nelson 81] examines this concept in detail.

systems on conventional uniprocessors) synchronous and asynchronous communication primitives are duals: there is a direct transformation from a system designed around one to a system designed around the other. Whether this duality extends to coordinated computing systems remains an open question.

Buffering One dimension of interprocess communication is the number of messages that can be pending at any time. In synchronous communication, each process has only a finite number of pending messages. Such systems have *bounded buffers.* In asynchronous communication, the system can allow an unlimited number of messages (*unbounded buffers*), provide a finite buffer that can be overwritten (*shared storage*), or halt a process that has created too many unresolved requests. These last two are also examples of systems with bounded buffers.

Information Flow The content of a message is its *information.* When processes communicate, information "flows" between them. This information can either flow from one process to the other (*unidirectional information flow*) or each process may transmit information to the other (*bidirectional information flow*). Bidirectional flow can be either simultaneous or delayed. With *simultaneous* flow, processes receive each other's communication at the same time. With *delayed* flow, first one process transfers information to the other, the recipient processes the request, and then sends an answer back to the original requester.* The classical procedure call is thus an example of bidirectional, delayed information flow. Systems with asynchronous communication invariably have only unidirectional information flow, as the sending process does not automatically receive a response.

We can imagine more complicated schemes, where information is transferred back and forth several times in a single communication. (This idea parallels virtual circuits in communications networks.) However, none of the systems we discuss support such a mechanism. One reason for this is that multiple exchanges can be imitated by repeated single exchanges.

Communication Control The dimension that has the largest variety of mechanisms is *communication control* — the rules for establishing communication. Most systems are concerned with focused communication—communications directed at particular recipient processes or "mailboxes." Some of these systems treat communicators *symmetrically*—each performs the same actions to achieve communication. However, most systems are *asymmetric.* These systems prescribe

* Our definitions of unidirectional and bidirectional information flow parallel similar concepts in the design of communication networks: a *simplex* connection transfers data in only one direction; a *half-duplex* connection, in both directions but not simultaneously (a version of our bidirectional delayed); and a *full-duplex* connection, in both directions simultaneously (similar to our bidirectional simultaneous).

a *caller-callee* relationship between communicators. In such an organization, one process makes a request to another. The called process can be *passive*, accepting all calls unconditionally, or it can be *active*, choosing between classes of requests. This selection takes many forms, which include named queues, guarded commands, pattern matching, time-outs, filters, and searches. Occasionally a system provides some of these mechanisms to the calling task.

Several heuristic systems use a pattern-invoking, broadcast form of communication. Here, the system conveys messages to their appropriate recipients, based on the contents of the messages and the interests of the recipients.

Communication Connection Communication can either be organized around a name external to the communicating processes (a *port*), a name associated with a particular process, or as a broadcast to interested tasks. Ports are most common in symmetric organizations. In asymmetric systems, communication is usually associated with either the called process as a whole (a *name*) or a particular label within the called process (an *entry*). In some heuristic systems, processes *broadcast* information. Recipients describe their interests by patterns and the system forwards appropriate messages to them.

Time A process that initiates communication may have to wait for its correspondent process. Such a process is blocked. Systems have various mechanisms for escaping this blocking. A few systems provide a mechanism for *timing-out* a blocked communication, permitting the process to register that the communication attempt failed. The most powerful such time-out mechanisms allow the programmer to specify an amount of time before the failure; weaker mechanisms provide only instantaneous time-out, where a process can check only if communication is immediately available. Although many models and languages do not support any time-based constructs, such constructs are vital for actual system implementations.

Fairness Intuitively, a *fair* system is one that gives each agent its rightful turn. Fairness is prominent in two places in coordinated computing—the fair allocation of computing resources and the fair allocation of communication opportunities among processes.

Formalizing the notion fairness is difficult. Oversimplifying, we say that there are three varieties of fairness: antifairness, weak fairness and strong fairness. An *antifair* system makes no commitments about the level of any process's service. In an antifair system, a process can make a request and be ignored for the remainder of the computation. In a *weakly fair* system, each process eventually gets its turn, although there is no limit on how long it might have to wait before being served. In a *strongly fair* system, processes with equivalent priorities should be served in the order of their requests. However, in a distributed system it is often difficult to establish the ordering of several concurrent events. Strong fairness is usually implemented by keeping an explicit queue of waiting requests.

Of course, all fairness criteria are modified by a model or language's explicit priority structure. If process A has higher priority than process B, then A may receive service arbitrarily more frequently than B in what is nevertheless a fair system.

The most common way to implement strong fairness is with queues. Frequently, every process entry in a strongly fair system has an associated queue. Processes accept requests on these entries in the queue order.

Failure One key issue of distributed computing is coping with failure. Several of the languages and models have features directed at dealing with particular kinds of failures. These mechanisms vary from language to language; they include time-outs, exception handlers, redundancy, atomic actions, and functional accuracy. *Time-outs* specify that a lack of response within a specific time period is to be treated as the failure of the correspondent process. In that case, a specified alternative action is to be executed. *Exception handlers* generalize this idea to other classes of failures, attaching programs to each variety of failure that the system can detect. *Redundancy* provides mechanisms for repeatedly attempting a fragile action in the hope that it will occasionally succeed. *Atomic actions* are a linguistic mechanism for encapsulating a group of more primitive actions, asserting that all are to fail if any fails. *Functional accuracy* embeds sufficient redundancy in the processes, programs, and data of a problem that even errorful intermediate results do not alter the system's overall performance.

Heuristic Mechanisms Several systems include heuristic mechanisms to aid in distributed control. These include atomic actions, pattern-directed invocation, and negotiation-based control. Part 4 discusses systems that focus on heuristic control issues.

Pragmatics Many of the proposals (particularly the languages) are intended as real programming tools. As such, they include features to ease the task of programming. These features include strong typing, symbolic tokens, and data abstraction. These constructs do not change the semantics of the underlying systems—whatever can be programmed with such aids can be programmed without them. However, there is considerable feeling in the programming language community that such features are essential to the design and construction of viable programming systems. When appropriate, we describe the pragmatic aspects of systems.

PROBLEMS

5-1 What other task domains could use a coordinated computing model or language?

5-2 Analyze a natural (human) organization, such as a company, school, or government, for the interactions described by our dimensions.

REFERENCES

[**Lauer 78**] Lauer, H. C., and R. M. Needham, "On the Duality of Operating Systems Structures," *Proc. 2d Int. Symp. Oper. Syst.*, IRIA (October 1978). Reprinted in *Operating Systems Review*, vol. 13, no. 2 (April 1979), pp. 3–19. This paper divides operating systems into two classes, those with a set of independent processes that communicate by messages and those with a set of processes that communicate with procedure calls and shared data. The paper asserts that a system organized by either method has an equivalent dual organized the other way. Furthermore, the dual is as efficient as the original.

[**Nelson 81**] Nelson, B. J., "Remote Procedure Call," Ph.D. dissertation, Carnegie-Mellon University, Pittsburgh (1981). Reprinted as Technical Report CSL-81-9, Xerox Palo Alto Research Center, Palo Alto, California. Nelson argues that remote procedure call is an appropriate basis for organizing distributed systems.